Simulation of Real-Time Scheduling Algorithms with Cache Effects

Maxime Chéramy, Pierre-Emmanuel Hladik, Anne-Marie Déplanche, Silvano Dal Zilio

Simulation of Real-Time Scheduling Algorithms with Cache Effects

Maxime Chéramy*, Pierre-Emmanuel Hladik*, Anne-Marie Déplanche† and Silvano Dal Zilio*
*CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
Univ de Toulouse, INSA, LAAS, F-31400 Toulouse, France
†IRCCyN UMR CNRS 6597, (Institut de Recherche en Communications et Cybernétique de Nantes), ECN,
1 rue de la Noe, BP92101, F-44321 Nantes cedex 3, France

***Abstract*—This article presents an extension of the simulator SimSo in order to integrate cache effects in the evaluation of real-time schedulers. A short presentation is done to expose the sources that give rise to temporal overheads in a real-time system and how to model and compute them in a scheduling simulation. We also present the benchmarks we used to validate these models and the tools to collect input data required for the simulation. A short example shows how the integration of cache models within the simulation opens new experimental perspectives.**

I. Introduction

The current trend in the embedded real-time systems is towards the use of multicore/multiprocessor architectures that offer enhanced computational capabilities compared to traditional single core ones. However, dealing with the concurrency of processing resources caused by parallel execution of programs in such platforms has led to new challenges. Consequently, since 2000, a plethora of papers have been published in the area of real-time multiprocessor scheduling giving rise to the design of numerous novel scheduling algorithms.

These scheduling algorithms were introduced together with analyses of their behavioral properties and schedulability capabilities. Such evaluations are conducted either in a mathematical way (by establishing and proving theorems) or empirically (by measuring some metrics on schedules of randomly generated tasksets). They aim at defining conditions guaranteeing schedulability, the percentage of schedulable tasksets, the number of task preemptions/migrations, the number of scheduling points, the algorithmic complexity, and so forth. Subsequently such results are used to compare the effectiveness of multiprocessor scheduling algorithms. However, such evaluations are commonly conducted on simple models, in particular for the hardware platform and operating system, that, at worst ignore or, at best take into account very approximately the actual overheads caused by operating system level mechanisms. Such overheads result in increased execution times for real-time tasks, thereby not only causing increased task response times but also influencing subsequent scheduling decisions in a correlated way. When analyzing scheduling algorithms, it is essential to be able to take them into account as precisely as possible so as to get valid results on their performance.

Our research work deals with the experimental study and evaluation of real-time multiprocessor scheduling algorithms. To this end, we have developed an open source software simulation tool called SimSo ("SImulation of Multiprocessor Scheduling with Overheads"), that is able to simulate the execution of tasks on a multiprocessor system according to the decisions of the scheduler. A cycle-accurate simulator or a real multiprocessor platform would give very precise measurements but it would also require to develop the scheduler in a low-level language, integrate it into an operating system and use concrete tasks. In contrast, SimSo is a simulator that is able to simulate a real-time system and to compute various metrics to be analyzed. But, neither a real implementation of the tasks nor an operating system are required, and extensive experiments can be easily conducted. Moreover, SimSo integrates behavioral models of hardware and software elements that impact the timing performances. As a consequence, it is able to take into account some of the overheads caused by the system level mechanisms such as: the computation of scheduling decisions, the operations of task context saving/loading, and the memory cache management. Such experimental simulations with SimSo enable to produce empirically results (rather than worst-case ones) and thus to bring out behavioral trends of scheduling algorithms.

The aim of this paper is to present how we have extended the models used to simulate the timing behavior of the tasks while taking into consideration the operating system and the hardware architecture. A particular attention will be given to memory cache effects. Indeed, additional information is required to estimate as accurately as possible and to include those overheads in our simulation approach. With this intention, we use models that come from the performance community to calculate cache miss rates and to estimate job execution times. However, the design of SimSo allows to integrate other models.

This paper is organized as follows. First, we present the main sources of temporal overheads to consider in a real-time scheduling simulation. In Section III, a short introduction to SimSo is done and the cache model and evaluation computational process are introduced. Section IV gives information on benchmarks and tools used to populate input data for the cache model. In Section V a short example illustrates the usage of SimSo to study the effect of caches as a function of offset and partitioning choices. Finally, in Section VI we give our conclusion.

II. Temporal overheads

Hereafter we examine major sources of overheads that may have an impact on the timing application behavior. Our objective was to integrate these aspects in our evaluations using simulation.

A. Scheduling overhead

The scheduler is in charge of allocating sufficient processing capacity to the tasks in order to meet all the timing constraints regarding their execution. For this, it has to solve two problems: i) an allocation problem, i.e. on which processor a task should execute[1]; ii) a priority problem, i.e. when and in what order it should execute each task. When considering online scheduling, such problems are solved by executing the scheduling algorithm at run-time. Whatever its type (partitioned, global, or hybrid, and time-triggered, or event-driven), the operations of the scheduler induce a run-time overhead that depends on the complexity of its decision rules for selecting the next tasks to execute. Depending on their implementation (on a dedicated processor, or distributed on the processors) and how kernel data structures are shared with possible contentions, the overhead may be incurred on one or more processors.

B. Context switching overhead

Each time a task begins or completes, is preempted or resumed, or carries on its execution on another processor, the context of the task must be saved or restored. In practice, the time penalty associated to a context switch is dependent of the hardware (processor registers, pipelines, memory mapping, etc.) and the operating system (update of internal states).

C. Cache overhead

Modern multiprocessor platforms usually use a hierarchical memory architecture with small and fast memories called caches, placed near the processors to alleviate the latencies of the slow central memory. In practice, the memory architecture may be composed of several levels of caches, from the level 1 cache (L1, the nearest memory from the processor) to levels 2 (L2) or 3 (L3).Their goal is to improve the overall performance by keeping in a fast memory the data that have a good chance of being reused. However, when multiple tasks execute and share the same caches (either concurrently or alternatively), the number of cache misses can increase and as a consequence also increase the computation time of the tasks. The situations we look at hereafter, aim at illustrating such phenomena.

Let us consider a task τ_a running on a processor π_j and another task τ_b that preempts τ_a. After the completion of τ_b, τ_a can be resumed on π_j. However, because τ_b executed on π_j, the instructions and data of τ_b can replace those of τ_a in the L1 cache. After the resumption, the task τ_a must wait for its instructions and data to be looked for in higher levels of caches or even in the central memory. The execution of τ_a is therefore delayed by the resulting cache misses that would not have occurred if the data had remained in the cache. A similar issue happens when a task migrates from one processor to another.

Moreover cache thrashing may take place with caches shared between processors. Thrashing occurs when the amount of cache desired by the tasks simultaneously executed exceeds the cache size. As a result, those tasks experience high cache miss rates since their instructions and data are frequently evicted from the cache before they can be reused. This, in turn, results in a severe degradation in task execution times.

[1]Note that some scheduling algorithms can change, at runtime, the processor that runs a task.

III. Simulation of the Cache Effects

The consideration of these overheads could supplement the evaluation of real-time scheduling algorithms. Therefore, we have integrated them in our simulation tool named SimSo [1]. The processing time taken by the scheduler and task context switching operations can be quite easily included in the simulation by adding (constant or state-dependent) temporal penalties to the handling of some specific simulation events (scheduler call, task start/resume, task completion/preemption). On the other hand, it is much more difficult to tackle the issue of cache related overheads. As mentioned above, they are strongly dependent of the data and instruction access patterns of the tasks. Thus they made us extend the inputs of SimSo and design cache models able to predict an estimation of the number of cache misses.

A. Execution Time Model in SimSo

SimSo is a simulator that allows to study real-time scheduling algorithms [1]. The purpose of the tool is not to simulate the real behavior of a specific system, but to evaluate empirically the behavior of the scheduling algorithms. More than 25 scheduling algorithms have been implemented so far.

In our previous work [1], we have shown that its design allows to simulate some temporal overheads: overheads related to the scheduling method, context save/load operations and the timer routines. In the simulation, these overheads do not increase the computation time of the jobs but keep the involved processor busy. Note that they can be configured by the user easily.

However, it is also possible to integrate some overheads in the simulation by affecting the simulated computation time of the jobs dynamically. SimSo offers the possibility to select the model used to determine the computation time of the jobs. More precisely, the models must be implemented in the simulation through the *Execution Time Model* (ETM) mechanism (Figure 1). The ETM is a single object that only interacts with the jobs in order to keep its independence from the rest of the simulation and to remain easily exchangeable by another one. Every time the state of a job changes, the ETM is informed so as to be able to determine the computation time of the job. The ETM provides to the jobs a lower bound of the remaining execution time. Once this time is equal to zero, the job ends.

Such an *Execution Time Model* has been implemented in order to simulate the cache effects. The following sections describe the additional data that were added to characterize the systems and the models that are implemented.

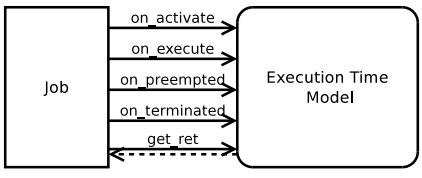


Fig. 1. Interface of the execution time model.

B. Additional characteristics

In order to compute execution time with cache, new task characteristics must be added to the classic task model. These characteristics (see Table I) come from the scientific domain of the hardware performance evaluation. Due to a lack of space, only the characteristics useful for the selected models are presented below. However, other metrics exist such as the Working Set Size [2] or the reuse distance [3].

TABLE I. ADDITIONAL TASK PROPERTIES

$base\_cpi$	mean number of cycles per instruction without considering memory access latency
n	total number of instructions to execute for each job
API	mean number of accesses to the memory per instruction
sdp	stack distance profile (see section III-C)
$CRPD$	fixed cache-related preemption delay
$CRMD$	fixed cache-related migration delay

These data are setup by the experimenter and can be either generated from real programs or artificially generated. It is important to remember that this will only help to build a memory behavior profile, but it cannot be used to reproduce the behavior of a specific program.

For each processor, a list of caches is provided. These caches are inclusive and can be shared among several processors. Table II sums up the characteristics of each cache.

TABLE II. CACHE PROPERTIES

name	name used in the cache hierarchy
$S_{\mathrm{L}x}$	size of the cache Lx expressed in cache lines
access time	time to reach this cache from the processor in cycles

C. Estimation of the cache effect

The penalties and the cache misses depend on the hypothesis on the architecture such as inclusivity, separate data and instruction caches, etc. Here, we assume that: the replacement policy is LRU; the caches are inclusive and hierarchical; the effects of the coherence protocol are negligible[2]; and only data caches are considered, the instruction cache is supposed modeled in the $base\_cpi$.

To update the progress of a job of a task τ, the ETM of SimSo evaluates the number of instructions n_τ executed by τ between two consecutive events of the simulator with

$$n_\tau = \frac{\Delta}{cpi_\tau} \tag{1}$$

where Δ is the elapsed time between the two events and cpi_τ the estimated average number of cycles needed to execute an instruction (CPI) of the task τ during this interval.

As the execution progress of a task is dependent of the cache consumption, the cpi_τ of a task τ is evaluated by considering the cache miss rates on each cache level and the penalties associated with a cache miss.

Under our cache hypotheses, the cpi_τ can be computed with

$$cpi_\tau = base\_cpi_\tau + API_\tau P_\tau \tag{2}$$

[2]The effects of coherence protocol increase with the number of cores, therefore, our model is limited to four cores.

where P_τ is the mean penalty associated to a cache miss. P_τ is defined by

$$P_\tau = mp_0 + \sum_{x=1}^{L} mr_{\tau,Lx} mp_{Lx} \tag{3}$$

where mp_{Lx} is the penalty of a miss on the cache level Lx, *i.e.* the difference between the time to access the cache level $L(x+1)$ and the time to access the level Lx, mp_0 is the penalty of any memory access, and $mr_{\tau,Lx}$ is the miss rate of τ for the cache level Lx.

An easy way to evaluate the miss rate of a task under LRU policy is to use the stack distance profiles (SDP), noted sdp_τ for the task τ. An SDP is the distribution of the number of unique cache lines accessed between two consecutive accesses to a same line [4]. An illustration of this distance is provided by Figure 2. Such metric can be captured for both fully-associative and N-way caches [5], [6]. For the LRU policy, the miss rate for a task τ in isolation is

$$mr_{\tau,Lx} = 1 - \sum_{i=0}^{S_{Lx}-1} sdp(i) \tag{4}$$

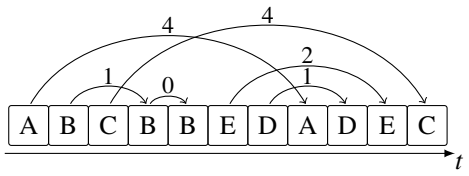


Fig. 2. Memory accesses sequence. A, B, C, D and E are cache lines and numbers indicate the stack distances.

Many works propose a way to compute $mr_{\tau,Lx}$ by considering the cache sharing between tasks on a multiprocessor architecture [3], [5], [6], etc. In a previous work [7], we have shown that the best compromise between computation performance and precision to compute the miss rate under cache sharing is given by the FOA algorithm [5]. This model considers the effect of the cache sharing as if the task is in isolation with a smaller cache size. This size is computed with the cache access frequency of each task. For the task τ the access frequency is

$$Af_\tau = \frac{API_\tau}{cpi\_alone_\tau} \tag{5}$$

where cpi_alone_τ is the CPI of τ executed in isolation (*i.e.* without other tasks). This value is simply computed with (2), (3) and (4). The virtual size of the cache for the task τ is given by

$$S_{\tau,Lx} = S_{Lx} \frac{Af_\tau}{\sum_{i=1}^{N_{\mathrm{L}x}} Af_i} \tag{6}$$

with N_{Lx} the number of concurrent tasks for the cache level Lx. To compute the cache miss rate, the equation (4) is used with the new estimated size $S_{\tau,Lx}$ instead of S_{Lx}. The Figure 3 represents the computation steps for the FOA algorithm.

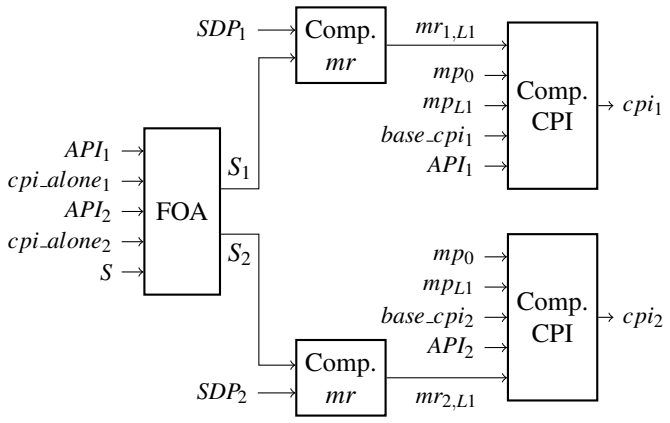


Fig. 3. Computation of the *cpi* with FOA for two tasks sharing a cache L1.

D. Preemption and migration costs

Regarding the cost induced by a preemption[3], several experiments are presented in [7] and show that it is difficult to predict it. Preemptive costs depend of the instant of the preemption, the state of the caches, the duration of the preemption, etc. However, the overhead due to preemption is small relatively to other durations. For all these reasons, we believe that a fixed penalty Cache Related Preemption Delay (CRPD) for each task is an acceptable alternative. The cost induced by migrations is considered exactly in the same way by adding a fixed penalty – Cache Related Migration Delay (CRMD) – when a task migrates.

IV. Tools

Before implementing cache models in SimSo, our concern was to evaluate their estimates on some effective task programs. We have thus collected a set of characteristics regarding the memory behavior of some tasks [7]. These studies allowed us to better understand the concept of SDP and to notice a large variety of possible behaviors. To that end, we have used a combination of several programs that comes from benchmarks used for performance evaluation and embedded systems. We had to develop some tools in order to measure the SDP.

In this part, we present the benchmarks we have used and the tools we have set up. We will then discuss the observed results and we will make some comments regarding the difficulties to experiment systems with the consideration of the caches.

A. Benchmarks

In order to conduct the experiments in [7], some programs were selected from the benchmarks MiBench [8] and Mälardalen [9]. These programs are representative of the kind of calculus used in embedded systems, contrary to those from SPEC CPU, which are commonly used to evaluate the performance of hardware architectures.

MiBench offers a large selection of concrete applications, however, some of them could no longer be compiled with recent compilers and some of them cannot run on the architecture simulator gem5 (see below). Moreover, some programs are too long, or at the opposite, do not make enough memory accesses to be interesting. With the aim of studying the cache effects, a dozen of programs from MiBench were kept (see [7] p. 164 for a complete list).

Regarding the programs from the Mälardalen suite, their main drawback is their very small computation time. Indeed, these programs were developed so as to easily compare WCET estimators. We have selected a dozen of them for which we have increased the computation time (usually by increasing the size of the input data). Note that some programs were compiled with different optimization options in order to see more profiles and to study this aspect on the SDP.

During our studies, we have run a lot of programs and we have collected their characteristics (SDP, API, number of instructions, base cpi). Among all the programs, we have kept approximatively 30 programs that show various behaviors when run alone or in concurrency. We now present how these characteristics can be collected.

B. Collecting data

The collection of the metrics is heavily based on gem5 [10], a low level architecture simulator. This tool was selected because it is still actively developed and possesses a very active community. The gem5 simulator is freely available and is sponsored, amongst others, by AMD, ARM, Intel, IBM or Sun. The use of an architecture simulator allows to control the hardware specifications of the system (e.g. number of cores, memory hierarchy and cache sizes).

To build the SDP, gem5 was modified in order to record the memory accesses. The file that contains the journal is then filtered to extract the sequence of the memory accesses made by the program on the first cache level. A second program takes as input this sequence and computes the SDP by simulating an LRU cache with infinite size. All these steps are automated by a Bash script (see Figure 4).

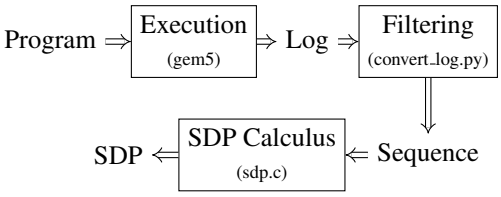
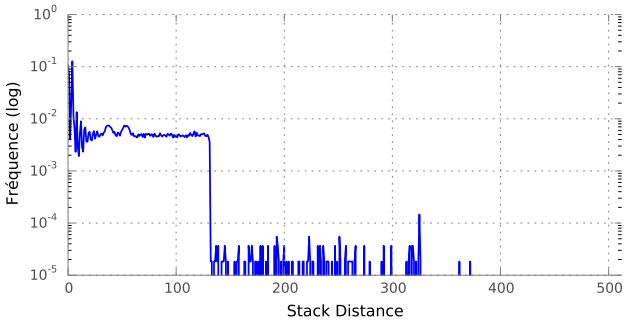


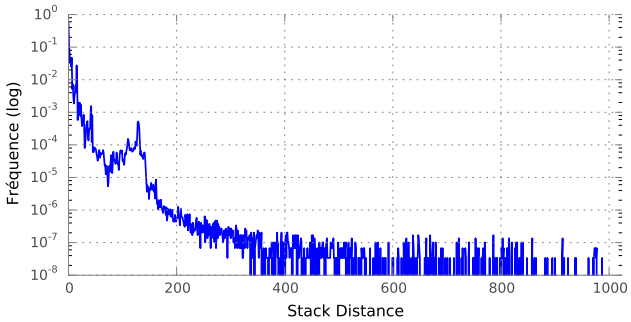
Fig. 4. Steps to generate the SDP of a program.

The study of the SDP of the various programs has shown a large variety of profiles. As an example, Figure 5 shows two SDP we have collected from the programs Blowfish and Say. The profile 5(b) is close to the usual assumption with a geometric distribution, whilst 5(a) shows two clear steps. Due to space reasons, we cannot expose here all the diversity of observed SDP, but we insist on the fact that this variety implies a large number of different behaviors. The main parameter to take into consideration is the size of the caches. However, when considering several programs running simultaneously or alternatively (with preemptions), other parameters also matter, such as the access frequency.

[3]Here, the cost of a preemption is the cost induced by the cache perturbation and not the cost of a context switch.



(a) Stack Distance Profile for *Blowfish*.



(b) Stack Distance Profile for *Say*.

Fig. 5. Example of two Stack Distance Profiles.

Finally, the CPI is the number of cycles needed to execute the program by gem5 divided by the number of instructions executed, and the API is the number of memory accesses divided by the number of instructions.

C. Generating artificial tasks

Some experiments evaluating the performance of schedulers use generators in order to simulate the cache consumption of the tasks. For instance, the evaluations conducted by Calendrino on LITMUS^{RT} are based on tasks that access the elements of an array [11]. The way these tasks access the array depends on the generator used. The first generator produces tasks that access the data of the array sequentially while the second generator produces tasks that access the data randomly.

We have reproduced these two generators and measured the SDP of the tasks. The first generator (sequential) induces more than 99% of the accesses done to a distance lower than 3. Regarding the second generator (random), 90% of the accesses are still done at a distance lower than 3 but the other accesses are uniformly distributed between 3 and the size of the array. We have also observed that the second generator refills the cache faster.

For all these reasons, we would like to underline the fact that it is difficult to evaluate a scheduling algorithm by taking into account the caches because the behaviors can be very wide and sensitive to the architecture characteristics. As a consequence, it would be interesting to study the bias introduced by the generators in the evaluations of some scheduling algorithms and better define the benchmarks to use.

V. Example

The integration of cache models within the simulation opens new experimental perspectives. In this section, we present a simple experimentation to demonstrate the possibility to conduct new kinds of evaluation on scheduling algorithms. More precisely, we study the impact on the partitioned EDF scheduler of the first task activation dates and of the way the tasks are allocated to the processors.

For this purpose, we have selected a set of 5 tasks for which the memory behavior has been collected from the execution of programs that come from the Mibench and Mälardalen benchmark suites. These tasks are periodic with implicit deadlines. Their total utilization is 1.35 (see Table III).

TABLE III. Task properties

Name	Period (ms)	ET[4] (ms)	#Instructions	API
Dijkstra-large	40	9	188 603 755	0.30
MATMULT-O2	30	10	47 922 711	0.37
COMPRESS-O2	20	5	46 771 323	0.28
Patricia	30	8	204 293 931	0.34
CNT-O2	10	3	143 775 036	0.08

The tasks run on two identical processors with a private cache L1 for each processor and a shared cache L2. The size of the L1 cache is 1Kio with an access time of 1 cycle, and the size of the L2 cache is 16Kio with an access time of 10 cycles. The access time of the main memory is 130 cycles (see Figure 6).

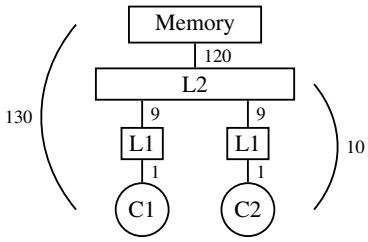


Fig. 6. Architecture of the memory

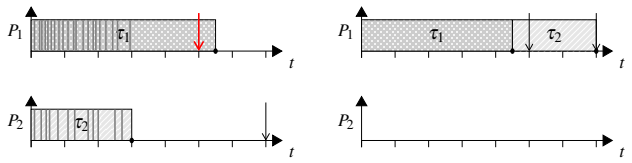
There is a total of 16 possible allocations of the tasks to the two processors with respect of the necessary and sufficient EDF schedulability condition ($\sum u_i \leq 1$ for each processor).

We focus on the way the use of the caches can be improved in order to reduce in practice the computation time of the tasks. For that, we consider two strategies. The first one consists in grouping on the same processor some tasks (typically those with a large cache occupancy) to avoid their simultaneous execution (see Figure 7). The second strategy consists in modifying the first activation dates of the tasks to take benefits from the idle times of the processors (see Figure 8).

Both strategies help reducing the effective system utilization[5]. The figure 9 shows the results for 3 different task allocations and for each configuration, 10 000 systems where generated with random activation dates.

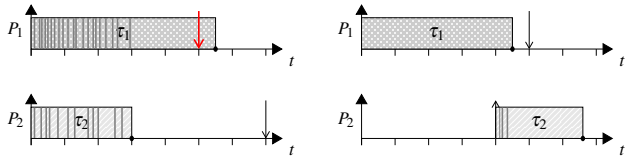
[4]WCET without considering the cache related penalties.

[5]The effective system utilization is the average time spent running tasks on the processors. It is similar to the utilization factor using the actual computation times instead of the WCET.



(a) One task per processor. (b) Tasks grouped on the same processor.

Fig. 7. How to avoid cache sharing by grouping the tasks on the same processor.



(a) Same activation date. (b) Shifted activation date.

Fig. 8. How to reduce cache sharing by delaying task releases.

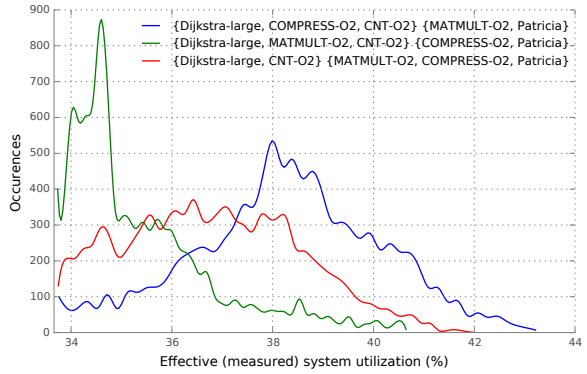


Fig. 9. Effective system utilization with various task to processor allocations and with randomly generated activation dates (5 tasks and 2 processors).

These results show that some choices, often ignored, may have some consequences on the task behaviours. SimSo, thanks to the integration of cache models, allows to operate new kinds of evaluations. This opens the possibility to compare scheduling algorithms on different criteria, which could be useful when they exhibit the same schedulability bounds.

VI. Conclusion

The purpose of the caches is to reduce the computation time of the programs. However, unless the system uses a cache partitioning mechanism, these caches are shared among the tasks. As a consequence, the gains from using the caches can be reduced when some tasks run in parallel or preempt each other. It also means that the decisions made by a real-time scheduler will have an impact on the actual computation time of the jobs.

This paper focuses on the way the cache effects are taken into consideration in the simulation to evaluate real-time schedulers. In particular, we have shown that SimSo offers the possibility to control the computation time of the jobs through the *Execution Time Model* mechanism in accordance with the cache model. Our goal was to reproduce, in the simulation, the impact of the caches on the computation time of the jobs and we decided to study these effects on the average case. However, it should also be possible to develop a new ETM that considers the impact of the caches in a worst-case manner.

In order to simulate the cache effects, we have studied the existing cache models that intend to estimate the execution time of the programs given the characteristics of the caches. We have selected a set of programs in order to study these models and we have decided to use the FOA model to reproduce the cache sharing effects. Our studies have shown that it was difficult to accurately estimate the cache related preemption delays, therefore, we decided to use fixed preemption and migration time penalties that increase the computation time of the jobs during the simulation. We have also integrated in the scheduling simulation some temporal overheads induced by the scheduler and the context switching mechanism with fixed time penalties.

Experiments to characterize the cache behaviors of a task show that the cache consumption profile can be very varied and needs dedicated tools and models to capture precisely their effects. The choice of benchmarks or algorithms to generate cache consumption can have a important bias on experimental results and need special attention.

Finally, the experimental evaluations on schedulers with the cache effects show that the simulation of the scheduling with cache models opens new perspectives.

References

[1] M. Chéramy, P.-E. Hladik, and A.-M. Déplanche, "SimSo: A simulation tool to evaluate real-time multiprocessor scheduling algorithms," in *Proceedings of the 5th WATERS*, 2014.

[2] A. Agarwal, J. Hennessy, and M. Horowitz, "An analytical cache model," *ACM Transactions on Computer Systems (TOCS)*, vol. 7, no. 2, pp. 184–215, 1989.

[3] E. Berg and E. Hagersten, "Statcache: a probabilistic approach to efficient and accurate data locality analysis," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2004.

[4] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, vol. 9, no. 2, 1970.

[5] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA)*, 2005.

[6] V. Babka, P. Libič, T. Martinec, and P. Tůma, "On the accuracy of cache sharing models," in *Proceedings of the third joint WOSP/SIPEW International Conference on Performance Engineering (ICPE)*, 2012.

[7] M. Chéramy, "Etude et évaluation de politiques d'ordonnancement temps réel multiprocesseur," Ph.D. dissertation, Université de Toulouse, 2014.

[8] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the IEEE International Workshop on Workload Characterization (WWC-4)*, 2001.

[9] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The mälardalen wcet benchmarks - past, present and future," in *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, 2010.

[10] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News*, 2011.

[11] J. M. Calandrino, "On the design and implementation of a cache-aware soft real-time scheduler for multicore platforms," Ph.D. dissertation, Chapel Hill, NC, USA, 2009, aAI3366308.